Reliable Data Transport Simulator

Sam Marsh

1 Stop-and-Wait

1.1 Sender

This stop-and-wait transport protocol is implemented using a finite-state machine, described in the `Sender` class by an enum `SenderState` containing values `WAIT_MSG` and `WAIT_ACK`. In the `WAIT_MSG` state, the sender waits for a message from the application layer. In the `WAIT_ACK` state, a packet is currently 'in-transit'. That is, the sender is waiting for acknowledgement from the receiver. An integer field `seq` tracks of the current packet. Since the stop-and-wait protocol is an alternating bit protocol, `seq` only holds either 0 or 1 (this is done by always setting its value modulo 2). In the `WAIT_MSG` state `seq` is used for the sequence number of the next message from the application layer.

This design, with an enum to represent the two major states, was chosen for simplicity of implementation. The stop-and-wait protocol can be represented as having only two states in which each network event is associated with a single state. Thus keeping track of the next sequence number, the in-flight packet and the current state was the most convenient approach. In a more complex protocol which involves multiple states where each state may react to several events and transition into several different possible states, it may be more effective to use a graph, where the nodes are states and the edges are actions caused by events.

1.2 Receiver

The receiver initialises its state with an expected sequence number of `expectedSeq=0` (representing the sequence number of the packet required next from the sender). Then it awaits a packet from the network. When a new packet is received, corruption and the sequence number is checked. If the packet is corrupt or has the wrong sequence number, the receiver acknowledges the previously received packet (0 if `expectedSeq` is 1, 1 if `expectedSeq` is 0). This causes the sender host to resend the expected packet after it times out. Otherwise if the received packet is valid, the payload is delivered up to the application layer and a response is sent to the sender host acknowledging the received packet, with acknowledgement number equal to `expectedSeq`. Finally `expectedSeq` is toggled $0 \to 1$, $1 \to 0$ and the receiver waits for the next packet.

2 Go-Back-N

The Stop-and-Wait protocol is a specific case of the Go-Back-N protocol described below. Indeed, the Stop-and-Wait protocol can be recovered from setting the `WINDOW_SIZE` in the following implementation to 1.

2.1 Sender

Upon receiving a new message from the application layer, a new packet is created and stored in the buffer with the message data and a sequence number of `nextSeqNum`. If the sequence number is within the window range, the packet is sent to the client. If the packet is first in the window, the timer is started. Finally, `nextSeqNum` is then incremented.

Upon receiving a new packet from the other host, the packet is first checked for corruption - if corrupt, it is ignored and no action is taken. Otherwise the acknowledgement number of the packet is extracted - all packets with `seq` below or equal have been successfully sent. In the initial implementation, this means the base could be immediately shifted up to one more than the received acknowledgement number. However this implementation involves buffering, so the actions taken at this point are different. If the packet is outside the window then it is ignored, since it is acknowledging an already-acknowledged packet. Otherwise, the running timer is stopped - unless the window is now empty it will soon be restarted. Then the window is stepped forward one slot at a time. At each step the end of the window is checked - if a packet exists there, it is transmitted. Finally unless the window is now empty, the timer is started again:

if the window is empty the timer should remain off because there are no in-flight packets, if non-empty the timer should be restarted to track the new base packet.

Upon timeouts, all in-window packets are re-sent and the timer is restarted.

The final implementation uses a single buffer for both storing the in-window packets and the buffered packets. The initial design instead used a separate queue of packets (`ArrayDeque<Packet>`) - upon a new acknowledgement packet being received and the window being moved, packets would be polled from the queue until either the queue was empty or the window was full. This was initially easier to implement, but maintaining two separate data structures soon became unwieldy and a number of bugs appeared. In addition, the queue had the potential to grow unboundedly large if messages flooded in from the application layer - resulting in possible crashes rather than dropping the messages. Even though this could have been prevented by tracking the queue size, in the a single array was a neater and less complex solution.

2.2 Receiver

The receiver stores two fields: `nextSeqNum`, holding the sequence number of the next packet required from the sender, and `sendPacket`, for acknowledging the packet with the highest in-order sequence number.

After initial setup (`nextSeqNum` = 1), the receiver has only one state - waiting for packets. If an incoming packet is corrupt or does not have the expected sequence number, `sendPacket` is (re-)transmitted to notify the sender of the required `seq`. Otherwise if the packet is valid, the payload is delivered to the application layer, then `sendPacket`'s acknowledgement number is set to the packet's sequence number and transmitted. Finally, `nextSeqNum` is incremented so that the receiver now waits for the correct next packet.

3 Checksum

In the `Checksum` class, methods for calculating the checksum of packet contents and validating a packet's checksum are available. The checksum calculation is:

`checksum` = `bitwise_not`(`seq#` + `ack#` + `integer value of each character of payload`)

`Checksum` also contains a method for checking if a packet is corrupt by calculating the checksum of the packet fields and comparing it to the packet's checksum field.